Constraint Propagation

This example presents a simple Constraint Propagation Engine. Constraint Propagation can be used to solve a variety of problems including some timetabling & scheduling problems as well as some involved with larger scale systems (knowledge representation, machine learning, language processing, etc). Constraint Propagation can be integrated with search or used on its own to solve some classic puzzle-type problems like Sudoku, Killer & Kakuro.

In this example we will consider Sudoku played out on a 4x4 grid (the principles are the same for a 9x9 grid but it is easier to examine the data with a 4x4 puzzle). Also (again for readability) the Sudoku cells will be labelled A..D rather than using numbers, eg...

A		B	
	C		A

start state

A	D	B	C
C	B	A	D
B	C	D	A
D	A	C	B

final state

For the sake of the following discussion we introduce a couple of computing terms. Sudoku cells will be represented as "nodes". The aim is to give each node a unique "label", at different stages in the processing any node may have a set of possible labels (all those labels that have not been excluded yet). Related nodes (those in the same row or column or 4-block) are grouped into "propagation sets".

We do not present an in-depth analysis of the strategy for solving Sudoku here. For the sake of this example we only consider one type of constraint... "related nodes may not have the same label".

representing the problem

Primarily we will be working with nodes & to enable this we need to give them names. Node names can be anything – here we name them with their row & column numbers so we can tell a nodes position from their name, ie:

n00	n01	n02	n03
n10	n11	n12	n13
n20	n21	n22	n23
n30	n31	n32	n33

As suggested above, the labels given to nodes will be one of A, B, C, & D. For convenience this label name info is held in a global variable...

```
(defparameter *all-labels* '(A B C D))
```

At the start of the puzzle, any nodes which have not been explicitly labelled will (by implication) have *all-labels* as their possible labelling. This information is held in a global structure called *labelling* which is an association list with one entry for each node. An entry (n30 A B C) would state that node "n30" could be labelled A, B or C.

Related nodes are grouped into sets each set is given a name & is specified something like... (r0 n00 n01 n02 n03) which states that set r0 (row-0) contains nodes n00, n01,

n02 & n03 (so they cannot be given the same label). The node set data is defined as follows...

```
(defparameter *node-sets*
  '(  ;; rows
    (r0 n00 n01 n02 n03)        (r1 n10 n11 n12 n13)
    (r2 n20 n21 n22 n23)        (r3 n30 n31 n32 n33)
      ;; columns
    (c0 n00 n10 n20 n30)        (c1 n01 n11 n21 n31)
    (c2 n02 n12 n22 n32)        (c3 n03 n13 n23 n33)
      ;; squares (t=top, b=bottom, l=left, r=right)
    (tl n00 n01 n10 n11)        (tr n02 n03 n12 n13)
    (bl n20 n21 n30 n31)        (br n22 n23 n32 n33)
    ))
```

The way Sudoku works, each node in the grid will be included in 3 sets (row, column & block). When a node N becomes uniquely labelled this information is propagated to all other nodes that share any of the sets with N.

The information about which nodes propagate to which other nodes can be derived from *node-set*. There are a couple of ways to do this. Here (for each node) we do this as follows...

step 1 – filter out any node-set data irrelevant to the current node (note use of mapcar #'rest to strip off the set name)

```
> (setf tmp (let ((node 'n00))
               (remove-if-not #'(lambda (nset) (member node nset))
                   (mapcar #'rest *node-sets*))))
((n00 n01 n02 n03) (n00 n10 n20 n30) (n00 n01 n10 n11))
```

step 2 – reduce this result using set union

```
> (setf tmp (reduce #'$+ tmp))
(n30 n20 n02 n03 n00 n01 n10 n11)
```

step 3 – remove the name of the node we are investigating

```
> (setf tmp (remove 'n00 tmp))
(n30 n20 n02 n03 n01 n10 n11)
```

step 4 – use mapcar (or something else suitable) to do steps 1-3 for all nodes (remembering to use the node name again to form a proper association list).

This propagation information is stored in a variable called *propagation-links* which ends up with the following structure...

```
((n31 n11 n01 n32 n33 n20 n21 n30)
 (n30 n10 n00 n32 n33 n20 n21 n31)
 (n21 n11 n01 n22 n23 n20 n30 n31)
 (n20 n10 n00 n22 n23 n21 n30 n31) ...)
```

starting things off

Sudoku puzzles are set up with a few key nodes given unique labels (see start state shown above). We will describe these using "problem set" data, we choose to represent these as a series of tuples like (#'node-set-label n00 A) where the first element is a function which sets a node's label, the second is a node name & the third is a label value. *problem-set1* (below) specifies the puzzle described earlier...

```
(defparameter *problem-set1*
  `((,#'node-set-label n00 A)
    (,#'node-set-label n02 B)
    (,#'node-set-label n21 C)
    (,#'node-set-label n23 A)
    ))
```

We manage these problem sets (and other parts of the CP process) using "process-request" queues[1].

process request queues

These kinds of queue go by various names – here we use "process request queue".

Normally, if part way through dealing with one function call we want some other function to be run, we simply call this other function (& suspend current operation until this other function finishes). But... sometimes we do not want to suspend current activity or do not want the second function to run until the first has finished.

We can deal with this by using a "process request queue" to queue our second function call until the first has finished. These queues are not difficult to set up or use, they are a good addition to the Lisp programmers cook-book.

How to build them...

```
(defparameter *pq* nil)             ; define a variable to hold the queue

(defun pq-enqueue (item)            ; queue a new item by putting
  (setf *pq* `(,@*pq* ,item)))      ; it on the back of the queue

(defun pq-dequeue ()                ; dequeue an item by pulling
  (pop *pq*))                       ; it off the front of the queue

(defun pq-isempty ()                ; is the queue empty?
  (not *pq*))

(defun pq-process-next ()
  (if *pq*                                ; if the queue has data
     (let ((next (pq-dequeue)))           ; pop the next item and...
       (apply (first next) (rest next))   ; initiate appropriate response
      )))
```

These queues allow us to queue (fns arg1 arg2...) forms. They are normally accompanied by some kind of iterative structure to call the process-next operation, eg...

```
(while *pq*
    (pq-process-next))
```

[1] "process-request queue" is our own term for this, other sources may use different names for these queues, they can be helpful in various situations.

process request queues for Sudoku

We don't need them!

Constraint Propagation works happily (though slightly differently) if we nest all the function calls rather than queuing them. So why bother?

Answer 1 – because it fits the logic of the way we would solve the problem better for many people.

Answer 2 – because it's a good place to introduce the concept.

The problem specification coded earlier is fed into the process request queue using the following function...

```
(defun setup-problem (problem-set)
  (dolist (p problem-set)
    (pq-enqueue p)
    ))
```

...and the call (run-problem *problem-set1*)

...the queued requests are dealt with in run-problem which (among other stuff) contains the following...

```
(defun run-problem (problem)
  ...
  (while *pq*
    (pq-process-next)
    )
  ...
  )
```

processing & propagating constraints

The example problem solution starts with a call to `(node-set-label n00 A)`. This does 2 things (check the code in the Lisp file)...

(i) changes the labelling for node n00 in the *labelling* association list so it is set to "A";
(ii) queues #'node-remove-label for all nodes in the propagation list for n00 so that "A" is removed from their *labelling*.

The second stage of processing deals with the queued #'node-remove-label requests. A call to `(node-remove-label N L)` does 2 things...

(i) it removes L from the list of possible *labelling* for node N;
(ii) if removing L from N reduces N to a unique labelling L2 then node-remove-label is called using L2 for all nodes in the propagation list for N.

In these ways, a request to remove a labelling for one nodes causes further requests to be issued and requests propagate between nodes. Eventually (assuming the problem is tightly enough specified and that we have adequate propagation rules) the problem reduces to a single consistent labelling. Importantly this happens without the kind of trial & error or expansion of state-space that happens with search. So, if problems are suitable for solving with CP it allows us to develop more efficient solution strategies.